Runtime Software Adaptation: Framework, Approaches, and Styles

Peyman Oreizy
Launch21
+1-425-442-9692
peymano@launch21.com

Nenad Medvidovic
University of Southern California
Computer Science Department
Los Angeles, CA 90089-0781
+1-213-740-5579
neno@usc.edu

Richard N. Taylor
University of California, Irvine
Institute for Software Research
Irvine, CA 92697-3455
+1-949-824-6429
taylor@ics.uci.edu

ABSTRACT
Our ICSE 1998 paper showed how an application can be adapted at runtime by manipulating its architectural model. In particular, our paper demonstrated the beneficial role of (1) software connectors in aiding runtime change, (2) an explicit architectural model fielded with the system and used as the basis for runtime change, and (3) architectural style in providing both structural and behavioral constraints over runtime change. This paper examines runtime evolution in the decade hence. A broad framework for studying and describing evolution is introduced that serves to unify the wide range of work now found in the field of dynamic software adaptation. This paper also looks to the future, identifying what we believe to be highly promising directions.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]; D.2.7 [**Distribution, Maintenance, and Enhancement**]

General Terms
Design

Keywords
Software adaptation; software evolution; software architecture; architectural styles; autonomic computing

1. INTRODUCTION
Runtime software adaptation and evolution concern changing a software system during its execution. Our work in runtime evolution (RE) was and is motivated by our society's increasing dependence on software-intensive systems and the real risks, costs, and inconvenience that their downtime presents. As we noted in our 1998 paper, "continuous availability is a critical requirement for an important class of software systems" [26]. Recently, it has become evident that this extends beyond the software that runs national power grids, global banking and financial systems, etc., and into commonplace systems such as:

- hosted email services (e.g., Google Gmail, Yahoo Mail, and Microsoft Hotmail), on which millions of people and businesses depend for communication, as updates are deployed to fix bugs, increase capacity, and provide new functionality;
- operating systems, where security patches that require a system reboot to install are not just inconvenient for end-users, but disruptive to mission-critical systems built atop these operating systems;
- consumer online banking systems, where a competitive analysis recently revealed that while many major U.S. banks had less than one hour of downtime over a two-month period, one of the nation's leading banks had over two days of downtime during the same period [28];
- cellular networks, as a recent outage of a popular network was traced to an issue with a "routine upgrade" [30].

Change is unavoidable in most systems: *intensive use breeds change*. Thus we need approaches that reduce, even eliminate, the costs and risks of evolving these systems without incurring downtime.

Our original paper and its follow-on journal version [27] were novel in their espousal of an architecture-based approach to RE. In particular, they demonstrated the beneficial role of: (1) software connectors in aiding runtime change; (2) an explicit architectural model fielded with the system and used as the basis for runtime change; and (3) architectural style in providing both structural and behavioral constraints over runtime change.

When we wrote our paper, research on the subject was scattered across a handful of workshop and conference tracks that broadly covered "software evolution", "configurable distributed systems", "programming languages", "operating systems", etc. At these venues, runtime evolution was one of many interesting aspects of a software system. In recent years however, interest in RE has grown substantially, especially in the area of architecture-based approaches at venues such as the *International Conference on Autonomic Computing (ICAC)* and the workshop series on *Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

Much progress has been made in the years since we wrote our paper. Research results and real-world experience demonstrate that software architecture can play a valuable role in achieving RE. In particular, the beneficial role of connectors, architectural style, and explicit architectural models come up repeatedly in this work (see sections 3 and 4). But research and experience also indicate that software architecture alone is insufficient in many situations. In real-world systems, it is common to see a variety of approaches used in concert:

- redundant or fault-tolerant hardware to cope with hardware failures;
- "hot pluggable" devices, in particular disk drives and memory chips, to add capacity or replace faulty units without power cycling a machine;
- the facilities of programming languages and their runtimes (e.g., Java's Virtual Machine and C#'s Common Language Runtime) to dynamically load, verify, and invoke code updates;
- system virtualization (e.g., VMware [1] and Xen [2]) to attain hardware fault isolation and improve resource utilization;
- tuning of operating system parameters to achieve optimal memory, CPU, and device utilization among application components.

It is evident that no single approach can encompass the others, and that no one system model can capture the diverse set of concerns necessary to effectively reason about and implement RE.

In our attempt to compare various approaches, we realized that no framework existed for effectively comparing approaches that operate at different levels of abstraction and utilize different system models, and that this was the crucial missing piece that prevented a holistic view of the problem. As a result, we developed a simple framework for making such comparisons, which we present in section 2. In section 3, we review progress to date (in both academia and practice), highlighting approaches that operate at different levels of abstraction that appear particularly effective at addressing aspects of RE. We conclude the paper in section 4 with some promising directions for future work gleaned from our study of the state-of-the-art and the state-of-the-practice.

2. A UNIFYING FRAMEWORK

To describe and illustrate our framework, we use a simple, contrived example in this section. Assume that we have a system that analyzes an infinite stream of images arriving from a deep space probe. The system continuously reads an image, applies several image processing algorithms to the image, and saves the images that are deemed "interesting".

Figure 1. Example data-flow model for image processing

A *software model* uses the principle of abstraction to hide certain details in order to highlight others. A system can be modeled in numerous ways, such as its structural architecture (Figure 2), its programming-language statements (Figure 4), the relationships between its data types (Figure 3), as data-flows through its subsystems (Figure 1), as the mapping between its virtual- and physical machines, etc. Note that we regard source code as a kind

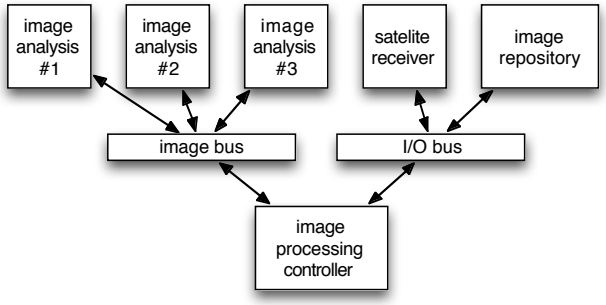

Figure 2. Example structural architecture model

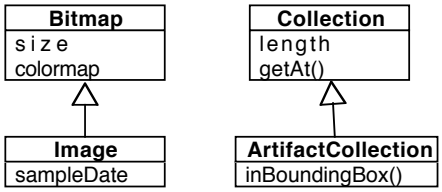

Figure 3. Example inheritance type hierarchy model

of system model. As we noted in Section 1, various models are useful for reasoning about and implementing runtime evolution.

One can imagine manipulating any one of these models to effect changes to a running system. Irrespective of the system model that is changed, an approach must address five aspects of change:

1. *Changes to the model's behavior*: How are such model changes represented, deployed, and applied? What aspects of the model's behavior can be changed? Can new behavior be added, can existing behaviors be replaced, or are we restricted to recombining existing behaviors? What assurances, if any, are provided that changes haven't been tampered with? In our example, the data flow model (Figure 1) could be altered to add new image processing algorithms, or the source code (Figure 4) can be "patched" to rewrite the implementation logic of one of the algorithms.
2. *Changes to the model's state*: How are state changes described? If the definition of a type changes, are existing instances altered? Are all instances updated simultaneously or lazily as they are accessed? Is the execution of the system stalled while state changes are made? In our example, the type inheritance hierarchy (Figure 3) could be altered to interpose a new type between the 'Bitmap' and 'Image' types.
3. *Changes to the execution context of the machine running our model*: A model is interpreted by a machine, e.g., an x86 processor, a Java virtual machine (JVM), a type inference engine, or a data flow engine.

As we make runtime changes to our model, we must be careful to not adversely affect the machine interpreting it. In our example, if we rewrite the x86 instructions of the process_image() function, the state of the x86 processor (registers, call stack, caches) and internal data structures of the operating system (process and thread state) may need to be updated as well.

4. *Asynchrony of change*: Applying a change at runtime is not instantaneous. It can take anywhere from several milliseconds to apply a patch to a small program's machine code to minutes or hours to change a large distributed system (e.g., due to communication latency or nodes being unavailable). Is the model's execution suspended during a change or does it continue to run in some capacity? If the latter, how does the approach deal with partially applied changes?
5. *Implementation probes*: Often, a change can only be applied when the running system satisfies a particular set of conditions. In our example, if we are patching the implementation of the analysis1() function and cannot do it correctly while the function is being executed, we would need a probe to tell us if the function is on the execution stack.

When we change a system model, the corresponding change must be made to any realization of the model, which (ultimately) includes the implementation. Any adjunct system models are likewise updated. Figure 5 depicts the processes involved. [1]

The lower half of the diagram, labeled "evolution management," focuses on the mechanisms used to change the application. System models are used as the basis for formulating and reasoning over runtime changes. Changes to these system models are reflected in modifications to the application's realization (at a lower level of abstraction), while ensuring that the model and the realization (which ultimately includes the implementation) are consistent with one another. Monitoring and evaluation services observe the application and its operating environment and feed information back to the diagram's upper half. The upper half of the diagram, labeled "adaptation management," describes the life-cycle of adaptation. The life-cycle can have humans in the loop or be fully autonomous. "Evaluate and monitor observations" refers to all forms of evaluating and observing an application's execution, including, at a minimum, performance monitoring, safety inspections, and constraint verification. "Plan changes" refers to the task of accepting the evaluations, defining an appropriate adaptation, and constructing a blueprint for executing that adaptation. "Deploy change descriptions" is the coordinated conveyance of change descriptions, components, and possibly new observers or evaluators to the implementation platform in the field. Conversely, deployment might also extract data, and possibly components, from the running application and convey them to some other point for analysis and optimization.

The system models of Figure 5 are related to one another, potentially in complex ways. In our example, when we change the data flow model, we are inducing changes in the system's source code and structural architecture. In the situation where a system model is realized in terms of another system model (e.g., a type hierarchy realized as source code), a change "trickles down" to the lower-level model. This can occur multiple times as a change trickles down to, say, the machine code. Likewise, changes to a system model can "percolate" up to higher-level models.

```
ArtifactCollection trackingArtifacts; // global state

void process_image(Image* image) {
   bool result1, result2, result3;
   result1 = analysis1(image, trackingArtifacts);
   result2 = analysis2(image);
   result3 = analysis3(image, trackingArtifacts);
   return result1 || result2 || result3;
}

image_processor() {
   ...
   do {
     event = wait_for_event();
     if (event.type == NewImageArrivedEvent) {
       bool shouldSave;
       Image* image = event.get_image();
       shouldSave = process_image(image);
       if (shouldSave) {
          ...
       }
     }
   } while (event.type != AbortEvent);
}
```

Figure 4. Pseudo-code for image processor

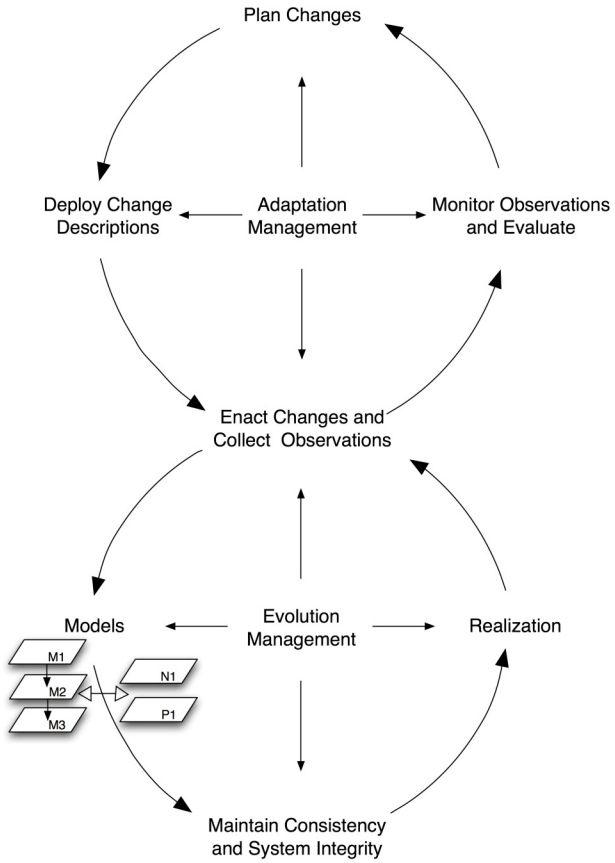

Figure 5. Processes involved in runtime adaptation

[1] Figure 5 is adapted from [27].

As a demonstration of the utility of our framework, assume that we have several important improvements that we would like to reason about and apply to the running system.

- *Changes to the model's behavior*: The technologies used by the deployed system would limit our options here, but assume that the deployed system offers both a binary-patching facility provided by the operating system that allows patching on a function-by-function basis (e.g., akin to Windows hotpatching [24]) and a software architecture-based adaptation infrastructure (akin to that describes in our original paper) that allows replacement of components. The consequences of this will be discussed shortly.
- *Changes to the program's state*: In our hypothetical case, data-flow analysis reveals that our changes only *reference* the "Image" and "ArtifactCollection" data types, and type analysis reveals that these two types have not been altered. Hence, these changes do not require updates to program state. If they did, we would need to look to approaches that support updates to state as well as behavior.

  As a result of this, we may prefer the binary patching facility since our changes can be applied in-place, avoiding the need to migrate state that would likely result if we used the component replacement approach.
- *Changes to the execution context of the machine running our program*: In our case, since this system uses an event-based implicit-invocation architectural style for triggering image analysis, we can deduce that it is safe to patch our code while the program is waiting for an event to arrive (at that point, we know that our code is not executing and that only the image_processor() function is on the program stack). We assume that waiting for this condition to occur is acceptable; if it is not, we would need to look to approaches that did not have this limitation.
- *Asynchrony of change*: Although we have identified a suitable condition that must be met for us to initiate our change (i.e., blocked on the arrival of an event), we must ensure that the condition isn't violated *during* the change. Hence, we decide to use an operating system mechanism that temporarily suspends the process while our binary patch is applied.

In spite of its simplicity, our example demonstrates the utility of our framework in two ways. (1) It allows us to reason about our change by combining observations gleaned from multiple levels of abstraction: data-flow and type analysis of the program's source code combined with observations about its architectural style guided us in choosing and applying the technique of binary patching to effect a safe runtime change. (2) It allowed us to compare two approaches for making changes (at the function-level versus structural software architecture-level) and choose the one that was best suited to the specific change we wanted to make. Of course, it is easy to reason about this trivial example in one's mind; a complex system would require tools that guided this effort and possibly automated some of its steps.

3. A LOOK BACK ON THE PAST DECADE

Several dynamic adaptation models have emerged in the past ten years. These models have tended to have an architectural focus and, as we will elaborate below, have had a number of shared characteristics. Additionally, a large number of research projects as well as several open source and commercial systems have taken on the different challenges of dynamic software evolution. These transcend not just software architecture, but also software engineering, confirming that software dynamism is a multi-faceted problem that can, and must, be approached from many directions. Finally, a number of conferences, symposia, and workshops have been initiated with software dynamism as a focal point. In this section, we will provide a brief overview of the accomplishments from the past decade. The ensuing discussion should not be viewed as a definitive survey of the state-of-the practice, but rather a collection of notable highlights.

3.1 Dynamic Adaptation Models

Several models of dynamic adaptation preceded our ICSE 1998 paper. These can be categorized into architectural style-based models, such as *CHAM* [18] and *graph grammars* [23], and architecture description language (ADL) based models, such as *Rapide* [21], *Darwin [22]*, and *Dynamic Wright [4]*. However, these models failed to gain wide adoption. There are two likely reasons: (1) the models were not accompanied by actual system-level facilities for dynamic evolution and (2) the type of dynamism they supported was in some ways overly constrained, such as only allowing an existing component to be replicated a certain number of times.

Our ICSE 1998 paper proposed an approach to runtime evolution intended to remedy these shortcomings. This approach was subsequently explicitly codified in the "*Figure 8*" model [27], a variation of which is depicted in Figure 5. The *Figure 8* argued that dynamic system evolution must be properly planned and carefully executed. Furthermore, it appropriately identified the ultimate target of dynamic evolution to be the *system* rather than one of the system's models (in Figure 5 this corresponds to the *realization* of models as the target of dynamic evolution). System models still play a critical role in that they are the drivers of evolution. This alleviated both of the above shortcomings of earlier dynamic adaptation models, while in the process also addressing the common problem of architectural erosion, where a system's architectural model and its implementation begin to diverge in significant ways.

A few years after this, another very similar architecture-based dynamic adaptation model emerged from the *Rainbow* project [16]. *Rainbow* also acknowledged the importance of maintaining the relationship between a system's architectural model and its implementation, performing on-the-fly analysis after the proposed modification to the model but before the system has been updated, and providing a style-based architecture implementation platform suitable for dynamic adaptation.

Recently, Kramer and Magee proposed a layered reference architecture for autonomous or self-managed systems [19]. Although it has been inspired by a particular class of autonomous systems—robots—this architecture is intended to be broadly applicable. The architecture's three layers are Component Control, Change Management, and Goal Management. *Component Control* contains the system's application-level functionality and supports the ability to add, remove, and reconnect components. This layer reports any events it is unable to process to the *Change Management* layer above, which in turn executes one of the pre-compiled plans to deal with a variety of situations the system may encounter. If none of the existing plans

can address the current situation, or a new system goal is introduced, then the top-most *Goal Management* layer is engaged to generate new plans.

A number of challenges are associated with this proposed reference architecture, some of which were recognized by Kramer and Magee [19]. Some of the challenges follow recurring themes in architecture-driven dynamic system adaptation. Those include maintaining the correspondence between architectural models and system implementations in order to ensure that architecture-based adaptations are properly effected, as well as providing the necessary runtime evolution facilities in the implementation infrastructure. Other challenges are specific to this particular reference architecture. A critical issue inherent in the architecture is efficiency: dynamic generation of plans can be a significant performance concern, especially when dealing with a system's (e.g., robot's) time-critical needs; this is further magnified if changes to the system's state cannot be treated in isolation and instead plans must be re-generated wholesale every time.

3.2 Research Projects

A large number of research projects have emerged over the past decade with some facet of dynamic adaptation at their core. Here we will briefly overview several such projects. We reiterate that the examples in this and the following subsection were selected because they facilitate interesting aspects of dynamism, and that the sections are not intended as definitive surveys.

Aura [15] is an architectural style and supporting middleware platform for dynamic pervasive systems, with a particular focus on context awareness and context switching. Aura supports software component mobility with the goal of ensuring required quality of service (QoS) levels. Its implementation infrastructure provides hooks for system self-monitoring, allowing the system to detect when requirements (e.g., response time) are not being met and, as a result, to deploy alternative configurations to support the task at hand.

MobiPads [9] is an example of a class of mobile middleware platforms. MobiPads support active deployment of middleware-level services for mobile computing. It monitors usage of middleware resources for specific QoS targets, and dynamically reconfigures them as required to optimize the QoS. On the other hand, MobiPads does not provide any application-level dynamic adaptation capabilities.

Siena [8] is a platform for deploying publish-subscribe systems across Internet-scale networks. Siena allows publishers (i.e., servers) to advertise their services, and subscribers (i.e., clients) to register for them. It then uses content-based routing to optimize the delivery of events from the servers to the appropriate clients. Siena supports dynamism inherently in that it allows clients and servers to enter and leave the system arbitrarily: the underlying infrastructure simply keeps track of the necessary routing information. Siena is also resilient to network failures, in that data can be re-routed dynamically. Given its focus, Siena's underlying publish-subscribe style, while explicit in the infrastructure, provides no additional guidance to application-level system designers.

Finally, recently a class of systems has emerged to support *grid computing [14]*. The term "grid application" refers to applications that have been adapted to use a distributed infrastructure (e.g., Globus [3]) and to run on "borrowed" hosts across a wide area network. Such applications are typically parallelized and written to accommodate the dynamic addition and removal of *physical* resources (e.g., PCs participating in the different @home [5, 20] networks). Since the underlying foundation is inherently unstable, software-level dynamism must be a top concern. At the same time, current grid systems are still very much script-driven, requiring system restarts for many types of adaptation. The focus of grid research has been on the infrastructure, with very little guidance given to grid application developers.

3.3 Commercial Solutions

A number of commercial solutions have also emerged over the past decade with varying degrees of dynamic adaptation capabilities. In this section, we select three of them as illustrative: a consumer electronics product family, a peer-to-peer voice-over-IP system, and an infrastructure for parallel processing of large data sets.

Koala [25] is an architecture-based technology for developing consumer electronics applications. Engineers at Philips developed Koala based on the Darwin ADL [22] and applied it initially to their large family of television sets. Koala allows an engineer to model and implement the software for, say, a TV set by composing existing components with pre-defined interfaces. Koala supports adaptation via two types of facilities: switches and options. Switches operate at the source code level (in the form of C `#ifdef`s), and require recompilation when setting values differently. On the other hand, options are stored in pre-programmed non-volatile memory. Options allow the use of a single ROM for multiple product types, but can handle only predefined runtime adaptations.

Skype [6] is a popular Internet telephony application built on a modified peer-to-peer (p2p) architecture. A new client logs onto the network via the Skype login server. After logging in, the client node is given the information about one of the supernodes. A Skype supernode handles all communication in a given portion of the network. Depending on network and usage characteristics, any node may be dynamically designated a supernode, or demoted from a supernode to a regular node, which results in a modification of the system's current topology. Furthermore, Skype inherits from the underlying architectural style, p2p, the ability to support runtime addition, removal, and even physical movement of hosts.

MapReduce [10] is Google's infrastructure for processing and generating large data sets. MapReduce users specify a map function which essentially divides a large data set into a number of subsets that can be processed in parallel. Users also specify a reduce function, which merges the intermediate data values as appropriate. Therefore, programs written in this style are amenable to automatic parallelization and processing on a large cluster of computers. MapReduce's runtime system handles the details of partitioning the input data, scheduling the program's execution across a set of machines, and managing the required inter-machine communication. MapReduce supports a limited (though critical to the intended domain) notion of dynamic adaptation, targeted at handling node failures: the runtime system automatically reroutes the data that was processed on a failed node to a live node.

3.4 Conference, Symposia, Workshops

Over the past decade, a number of conference, symposia, and workshops have cropped up that deal with different facets of software dynamism. These can be categorized into events whose

primary purpose is dissemination of ideas pertaining to dynamism, and those that embrace dynamism as a means of addressing other problems. We will discuss some examples of each category. We will conclude the section with a brief view to the role dynamism has played in the mainstream software engineering conferences, including ICSE.

3.4.1 Dynamism as a Primary Focus

The *International Conference on Autonomic Computing (ICAC)* has assumed the leading role in dealing with software dynamism across several computer science disciplines: AI, software engineering, programming languages, databases, HCI, mobile and pervasive computing, robotics, operating systems, networking, distributed systems, embedded systems, and even biology. The conference emerged several years ago as a direct outcome of IBM's autonomic computing initiative. The motivation behind it was the recognition that the increasing complexity of constructing, integrating, and managing software systems has frequently overwhelmed the capabilities of not just software engineers, but also system administrators. It is argued that the only viable long-term alternative to the current state-of-the-practice is to advance the field of autonomic computing, i.e., to create computer systems that manage themselves in accordance with high-level guidance from humans. The ICAC manifesto states that "meeting the grand challenge of autonomic computing requires scientific and technological advances in a wide variety of fields, and new architectures that support effective integration of the constituent technologies".[2] Over the past four years, ICAC has gathered researchers from many traditional computer science areas. It is still early to judge whether a unified, cross-disciplinary vision of autonomic computing has begun to emerge, but the amount of interest and research activity (also reflected in the recent formation of the *Autonomic Computing Workshop* and the *Conference on Human Impact and Application of Autonomic Computing Systems*) is heartening.

The workshop series on *Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* has been the software-engineering community's "answer" to ICAC. SEAMS has tried to consolidate a number of software engineering workshops that have emerged over the past decade to deal with various aspects of dynamic software adaptation, including the *Workshop on Self-Healing Systems (WOSS)*, *Workshop on Design and Evolution of Autonomic Application Software (DEAS)*, as well as the more broadly scoped *Workshop on Architecting Dependable Systems (WADS)* and *International Workshop on Principles of Software Evolution (IWPSE)*. The SEAMS organizers explicitly state that they are "attempting to consolidate interest in the ICSE and FSE software engineering communities on autonomic, self-managing, self-healing, self-optimizing, self-configuring, and self-adaptive systems".[3] While the stated goal of SEAMS is to bring together researchers and practitioners from diverse computer science areas to discuss the fundamental principles, state of the art, and critical challenges of self-adaptive and self-managing systems, its specific focus is on the software engineering aspects of self-adaptation and self-management: methods, architectures, algorithms, techniques, and tools that can be used to support software development in such systems.

The growing research activity in this area recently resulted in a *Dagstuhl Seminar on Software Engineering for Self-Adaptive Systems*. The motivating observation for organizing the seminar was that, while self-adaptation has been studied across many disciplines of computer science, software engineering is uniquely positioned to provide a platform for consolidating these results in that the common element that enables the provision of self-adaptation across all these areas is software. The objective, therefore, is to try and energize the software engineering community to devise a comprehensive, broadly applicable solution to self-adaptation.

3.4.2 Dynamism as a Means or By-Product

A number of additional conferences and symposia have dealt with dynamic software adaptation as either a means or by-product of achieving a related objective. For example, *MobiCom* is a conference dedicated to addressing research challenges in the areas of mobile computing as well as wireless and mobile networking. While many of the problems MobiCom tries to address are very low level and, on the surface, have little to do with software engineering (e.g., protocols for software radios or techniques for dynamic spectrum use), certain aspects of mobility will inherently involve the dynamic adaptation of the software, both at the system level and at the application level.

PerCom is a conference dedicated to the emerging area of pervasive computing and communications. It is explicitly aimed at providing a "platform and paradigm for all the time, everywhere services".[4] The conference is seen as a natural outcome of the advances in wireless networks, mobile computing, sensor networks, distributed computing, and agent technologies. Several of its areas of interest (e.g., wearable computers, pervasive computing architectures, context-aware computing, and autonomic computing) will inherently have to deal with on-the-fly adaptation of the underlying software.

The *Working Conference on Component Deployment (CD)* is an event targeted specifically at the issues dealing with software system deployment in distributed (possibly pervasive and mobile) environments. Software deployment is a dynamic activity in that software is relocated from a source host to a set of target hosts, although the system's initial deployment usually involves the transfer of inactive, stateless modules. If, however, deployment takes place during system runtime (in which case, this is, in fact, *re*deployment), then it is an instance of dynamic system adaptation. A number of approaches have been proposed at CD for dealing with this variation on dynamic adaptation.

Another specialized, long-running conference series that deals with issues pertaining to dynamic software adaptation is *Middleware*. Middleware is scoped much more broadly and deals with the provision of all types of enabling services for effective distributed computing. However, a predominant number of middleware platforms support at least some form of dynamism (e.g., dynamic discovery, insertion, and/or invocation of components in CORBA). While usually highly specialized, these techniques can and do inform software engineers interested in studying software system dynamism.

3.4.3 Dynamism in Our Flagship Conferences

The above discussion, while partial, provides ample evidence of the growing interest in dynamic adaptation, not just within

[2] http://www.caip.rutgers.edu/~parashar/ac2004/organization.html

[3] http://www.seams2007.cs.uvic.ca/

[4] http://www4.comp.polyu.edu.hk/~percom08/

software engineering, but also across many other areas. Yet, even a cursory look at the proceedings of the major software engineering conferences (ICSE, FSE, ASE) paints a curious picture: since 1998, there has been a smattering of papers dealing with dynamic adaptation; there are no technical paper sessions or panels dedicated to dynamism. A similar trend can be seen if we look at the major software architecture venues (e.g., WICSA). Given the proliferation of ICSE and FSE workshops dealing with this subject during this same period, this would seem to suggest that, as far as the software engineering research community is concerned, the problem of dynamic software evolution is a fascinating topic for discussion, but not (yet) worthy of concerted focus in our flagship conferences. This is disappointing. We hope that our paper being awarded the *Most Influential Paper of ICSE '98* is a harbinger of change.

4. PROMISING DIRECTIONS FOR FUTURE WORK: ARCHITECTURAL STYLES

The framework introduced in Section 2 presented a general model for characterizing and achieving dynamic adaptation. It discussed the range of issues that must be addressed when a system is modified. Progress over the past decade, presented in the preceding section, covered approaches that span a gamut of techniques within that general framework. Here we focus on a particular but broad approach that we believe holds the greatest promise for achieving highly adaptable systems in the future. The premise is quite simple: build systems in a manner that makes adaptation easier than otherwise. The guidelines that characterize the design of such systems – indeed, the constraints that such systems obey – constitute architectural styles.

Architectural styles are named collections of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system [32]. Here, the beneficial quality sought is dynamic adaptability.

4.1 Leverage Points: Making Adaptation Easier

The central notions of architectural styles that have been successful in facilitating adaptation, and which are essential in new styles that are designed to support adaptation, are:

- Making the parts subject to change identifiable and manipulable
- Controlling interaction with parts subject to change
- Managing state

Not surprisingly, these stylistic notions encompass key elements of the adaptation framework from Section 2. We describe these notions briefly, then discuss how they can be realized concretely and are exemplified in some specific and well-known architectural styles.

4.1.1 Identifying the elements subject to change

Obviously for an element of a system to be subject to replacement that element must be identifiable. More usefully, the lowest level at which an entity can be (potentially) manipulated is the lowest level at which it can be specifically identified. If a model element *x* is identifiable at abstraction level *n*, but is "translated away" to anonymously become part of a larger realization *y* at level *n-1*, then only at level n (or possibly, higher) can manipulation of *x* be discussed. This, of course, is the essence of interpreted systems: an entity which is named at level *n* may be manipulated by an interpretive scheme at level *n-1*. If however, the entity is translated to another representation level *n-1* such that it loses its identity, specific manipulation of that entity is no longer possible at level *n-1* or below.

While identifying the element to be changed is necessary, supporting change is greatly aided by promoting its "encapsulation". We use the word in its broadest sense here; the degree to which an entity can be severed from its surroundings determines how easy the element is, for instance, to replace. Consider, for example, a function performed by a few lines of code. If all the code is nameable, contiguous, has a single point of entry and a single point of exit, obtains its values from a single defined location and leaves it results in a single defined location, then excising that function and replacing it with another is straightforward. If however the lines of code are scattered through other code, has multiple entry points, communicates with other portions of code through dynamic storage, and so forth, manipulation of the function becomes more difficult.

From the standpoint of an architectural style, to the extent that a style fosters encapsulation of systems elements – such as by requiring all computation to be located in named components and all communication between components to occur through explicit connectors – *and* those encapsulations are preserved through the levels of abstraction down to executable code, then that style promotes adaptation.

4.1.2 Controlling interaction

Just as identifying and encapsulating a computational element is an aid in promoting adaptation, so is the identification and encapsulation of how an element communicates with other elements. If connections between elements are explicit and maintained as explicit entities across the levels of abstraction down to code, then adaptation is promoted, for the same reasons as described immediately above.

For example, if two components communicate only by exchanging explicit messages – identifiable bags of bits – then changing out one of those components is facilitated since the protocol of interaction is more readily identified across the layers of abstraction. Conversely, if two components communicate through shared memory (e.g., global variables), then identifying the precise means of communication is much more difficult.

4.1.3 Managing state

When a computational element or a communication element is changed out of an application, the state of that element must also be addressed. Components may have internal state; connectors may have buffers full of messages. Successful runtime adaptation may well require that the new, replacement part be initialized with part or all of the state held by the now-replaced element. One simple strategy is to require (as a stylistic constraint) that all components present an interface that forces the component to checkpoint its state externally, and another interface that causes the component to initialize itself from that external store [34]. A more interesting strategy is to require components to always maintain "their" state externally – a strategy brilliantly exploited by the REST style [13] and discussed below.

Managing state also includes establishing or identifying times of quiescence, times at which the execution state is meaningfully subject to change. A style could be imposed, for instance, which

guarantees a useful state of quiescence upon completion of a call to a designated interface.

4.2 Supporting the Leverage Points

The styles discussed below use a variety of techniques to promote adaptation, but two general strategies are apparent: delay bindings and use explicit events/messages in communication.

In its most common usage, delayed binding means that an entity is not linked, or "bound", to its usage context until the time during execution when the computation cannot proceed further in absence of a specific linkage. The concept can be applied at higher levels in the abstraction hierarchy than execution, however. The essence is that entities retain their identification and as needed become associated with a usage context. A plug-in component, for instance, may remain unbound to a master application until, e.g. application initialization, or even until the point at which the plug-in would be called to perform some service. More generally, components potentially may not be loaded into memory until the time they are needed. Clients may only be bound to a particular server once a load balancer determines which server out of some equivalence class has the best chance of quickly attending to the client. Messages may be delayed in binding to a specific destination, as they are routed across intermediaries. Delaying of bindings means that before bindings are wired in, adaptation can come "for free", with respect to that binding context.

Use of explicit, or "first-class" events/messages in communication facilitates adaptation in two main ways. First, such communication implies a strong decoupling of the components engaged in the communication: no direct procedure calls are involved, no shared memory, no tight bindings. Second, first-class events are encapsulated entities, so they may be examined, logged, or manipulated by third-parties to serve various adaptation needs. They may have associated meta-data (as occurs in HTTP/1.1 for example) that enable reasoning about what kinds of processing services they may require upon reaching their destination.

4.3 Styles That Make Adaptation Easier: Past, Present, and Future

Making life easier for oneself through use of architectural styles is a long-established practice of developers. We describe here several styles that have been used to support adaptation, and highlight a few styles that show great promise for the future. A summary of these styles' characteristics that particularly facilitate dynamic adaptation is provided in Table 1.

Proto-runtime evolution: Pipe and Filter. The pipe and filter style exemplifies explicit, severable components, explicit connectors, the use of common interfaces, and standardization on the root type of all messages (viz., ASCII streams). Creating a new pipe and filter application from an existing collection of filters is trivial. What pipe and filter lacks for supporting adaptation, however, is a mechanism for evolution management (the run-time change process) – for example the dynamic rerouting of a pipe from feeding one filter to feeding another.

Dynamic pipe and filter: Weaves. The Weaves system [17] exploited the strengths of pipe and filter and supplied the key missing elements for supporting run-time adaptation. Weaves provided for the dynamic rewiring of pipes through a combination of explicit buffering of messages and flow control mechanisms. In particular, when a pipe was detached from a consuming filter, the buffer within the pipe would accommodate messages up to its limit, but as the buffer's limit was approached the pipe would invoke a standard interface on the producing filter to retard or inhibit the production of more messages until the rewiring was complete and the buffer regained capacity for handling additional messages.

Events and notifications: Field and Publish-Subscribe. The Field software development environment [29] was created with the express intent of supporting dynamism: independent components (Unix tools) were invoked in response to receipt of explicit events routed through a centralized message connector. The set of tools that would respond to a particular event were not pre-specified and tools could be added to deleted to the set dynamically. Efficiencies in the process could be introduced through declaration of the types of events a tool would wish to hear about. This mechanism is conceptually that of publish-subscribe. Pub-sub, however, can be applied within a single application and need not involve explicit discreet events, but can be implemented by procedure calls. The degree of adaptivity achieved in such a choice is limited, however, to encompass the set of procedures for which a referent can be made available. Numerous pub-sub mechanisms have appeared over the past fifteen years (such as Siena [8], mentioned earlier in Section 3.2), enabling the technique to be effectively applied in a wide variety of circumstances [12].

Event-based components and connectors: C2. The architectural style used within our original paper was C2 [31]. C2 employed the event notification mechanism in a disciplined manner and combined it with other techniques designed to facilitate adaptation: components are readily severable from their usage context since the *only* way they may communicate with each other is through explicit messages routed by first-class connectors. C2's layering and visibility rules further promoted adaptivity, since they limited the latent or implicit dependencies that arise from knowing which component is going to service a particular event. The C2 style, however, makes no particular provision for managing state as a component is replaced.

Dynamism through replication: Tile Style. A radically different approach to adaptivity is found in the tile style. The tile style [7] provides a software architecture-based solution for computing NP-complete problems non-deterministically by harnessing the power of many networked computers simultaneously. The style is based on an underlying model of molecular self-assembly, which results in solutions known as *tile systems* [33]. Since the objective of the tile style is to produce software systems that can compute on public networks, the key intended properties of the style are *privacy* of data and computation, *robustness* in the face of adversary attacks and node failures, as well as *scalability* to very large problems and networks. The tile style achieves this by leveraging the property that all NP-complete problems can be transformed into one another. Therefore, a tile system that has been proven to solve a known NP-complete problem is used as a starting point. For each tile in this system, the corresponding tile-style architecture dynamically (1) deploys a simple tile component on a single machine in a network of willing participants, (2) connects that component with other components on the network as appropriate to effect the desired architectural configuration, and then (3) replicates this component two additional times in order to parallelize the computation. This three-step process is repeated for each of the replicas (thus resulting in 2^n dynamically generated

copies of the architecture), until either a solution is found or a pre-determined probability threshold that no solution exists for the given problem is reached. As a result of such massive replication, failures or security breaches of even large portions of a network (e.g., 10%) are withstood relatively easily.

Externalization of state: REpresentational State Transfer (REST). Arguably the most demonstrably successful architectural style in supporting runtime evolution of large-scale applications is REST [13]. REST is the underlying style of the modern World Wide Web, an application so dynamic that no representation of the architecture "now" is possible. The Web is continually changing as clients, servers, proxies, and gateways come and go with staggering rapidity. Designed to support the network exchange of hypermedia documents while preserving component independence (integration scaling in the face of agency borders) and minimizing latency (performance scaling), REST can be summarized as six simple principles:

1. The key abstraction of information is a resource, named by an URL
2. The representation of a resource is a sequence of bytes plus metadata to describe those bytes
3. All interactions are context-free
4. Only a few primitive operations are available
5. Idempotent operations and representation metadata are encouraged in support of caching
6. The presence of intermediaries is promoted

Each of the six principles contributes to the scaling, diversity, and ease of adaptation within the Web. URLs are an anarchic, decoupled namespace with no central authority and each Web server may support whatever URLs it chooses and assign whatever meaning it deems appropriate to each URL (resource). The freedom to introduce server-specific namespaces (URLs) and resources is partially responsible for the outpouring of innovative Web services and applications.

Analyzed in terms of principles of adaptation, REST principle #3, "all interactions are context-free" is the most critical. Sometimes

Change Aspect / Arch. Style	*Update behavior*	*Update state*	*Update execution context*	*Asynchrony of change*	*Implementation probes*
MapReduce	data from failed nodes can be dynamically resubmitted to live spares for processing		execution contexts of failed nodes are reassigned to other nodes		status updates for displaying progress; server logs
Pub-Sub	publishers and subscribers can join and depart			varies; pub-sub bus can buffer events, etc.	subscribers can act as probes; pub-sub bus can provide probes
Weaves	changes to dataflow model mapped to implementation; can add/remove components and connectors; can change flow of data			flow control in connectors (buffering, rate mgmt, etc.)	probe infrastructure for displaying behavioral and performance data
C2	changes to architectural model mapped to implementation; can add/remove components and connectors; can change topology		various techniques resting on explicit connectors (e.g., multi-versioning connectors)	all communication is asynchronous, so can exploit control in connectors	message probes on connectors
Tile Style	discovery and recruitment of new hardware nodes on which components type are to be replicated	exchange (partial) maps of local neighborhoods across neighbor hardware nodes	systems are intended to "borrow" cycles from host machines	apply random graph walk to ensure load balancing	nodes checked regularly for liveness during recruiting and replication
REST	stateless http servers can be restarted for updates; database servers updated using vendor-specific techniques	state is externalized: all messages carry state and may be inspected; http server is stateless; application state stored in database servers	before update, "drain" in-process requests and refuse new requests	various techniques, e.g., shift load to ½ nodes, update idle nodes, shift load to updated nodes, update remaining nodes	server logs; query state in database
CREST	stateless servers (peers) may offer URL-specific interpreters; nominal behavior encapsulated in computations that are transmitted	all aspects of a computation's state made explicit and externalized	fully included within the computations (i.e., continuations) exchanged between peers	Same as REST	server logs; computations are explicit and transmitted, hence may be examined by intermediaries

Table 1: Summary of dynamic adaptation features of several architectural styles discussed in the paper.

phrased rather misleadingly as "the protocol is stateless", the principle demands that all state be externalized. A message (to a server, for example) must carry whatever state with it is necessary for that server to be able to process it, without recourse to any prior history of interaction. Principle #2, which includes the use of meta-data to describe the content in a message, further promotes adaptation as components may inspect a message and determine how to handle it based upon that meta-data. Principle #4 keeps the barrier for introducing new processing components low.

In all, REST represents a remarkably insightful blend of architectural principles to support dynamism within the domain of distributed hypermedia.

One look to the future: CREST. The one aspect of adaptation identified in the framework presented earlier that is poorly addressed by all of the above styles – if addressed at all – is execution context. The concept of application quiescence is important as it recognizes that a component may be "part way" through some computation, and delay in adaptation may be warranted to allow that computation to finish. REST externalizes state in communication, but makes a fairly sharp distinction between clients and servers, wherein computation is confined to servers; clients (browsers) just present data. Of course recent experience with JavaScript and AJAX reveals an increasing presence of computation on clients – but in a way not predicted or leveraged by REST. The opportunity is to take a bold step forward, supplanting REST with what may represent the ultimate in adaptivity: make the fundamental unit of exchange on the Web *computations*, not simply representations [11]. The resulting style, Computational REST, or CREST, affords the prospect of bringing the degree of adaptivity seen in the current Web to all manner of multi-party distributed computation.

In CREST, the key abstraction of computation is a resource, named by a URL. Any computation that can be named can be a resource: e.g., word processing, image manipulation, a temporal service (such as "tomorrow's weather in Boston"), a generated collection of other resources, a simulation of an object. The representation of a resource is a *program* to be executed plus descriptive metadata describing the program. Analogously to REST, in CREST all computations are context-free. This is not to imply that applications are without state, but that each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it. Prior representations can be used to transfer state between computations; for example, a *continuation* (representation) provided earlier by a resource can be used to resume a computation at a later time merely by presenting that continuation. In this manner all aspects of application state are externalized, yielding adaptivity.

5. CONCLUSION: A CALL TO ACTION

Dynamic adaptation of software systems is becoming recognized as a critical capability in many application domains, research projects, commercial systems, as well as across a large number of computer science disciplines. It is also becoming increasingly clear that software engineering is uniquely positioned as *the* discipline that can provide the needed know-how for addressing the many significant challenges associated with dynamic adaptation. This is an opportunity our community must embrace! Doing so will, however, require an expansion of our collective focus, and perhaps even a change in our collective mindset.

The authors of this paper certainly acknowledge that applying formalisms to analyze software systems' models or their particular phenomena, analyzing and testing already built systems, studying programmer productivity and program quality, measuring and improving development processes, and so on are all important aspects of our discipline and require continued careful study. We refer to these areas loosely as *sciences of software analysis*. Our community's work in this arena continues to have a broad and lasting influence.

At the same time, we argue that a much more concerted effort than has been made thus far is needed to develop the corresponding *sciences of software synthesis*. This will include providing better methods, techniques, and tools for

- designing large and complex software systems – *a science of design*;
- implementing those systems in a manner that encourages preservation of the principal design decisions (as opposed to encouraging their violation as is frequently the case today) – *a science of realization*;
- enabling their dynamic adaptability in a variety of situations, especially those that are not foreseeable at design time – a *science of dynamic adaptation*; and
- taking advantage of the domain characteristics that are particularly amenable to effective design, realization, and dynamic adaptation – *a science of domain-specific software engineering*.

What we are advocating is a huge undertaking to be sure. However, it is not one we can afford to shy away from. Our community must find a way of encouraging our best and brightest to consider the problems of software synthesis at least as worthy of their attention as those of software analysis. We believe that only with such a focused community-wide effort will software engineering be able to realize its potential for leadership and critical impact in the area of dynamic adaptation.

6. ACKNOWLEDGMENTS

This work sponsored in part by NSF Grants CNS-0438996 and ITR-0312780 for which the authors are deeply grateful. The authors wish to thank Kokichi Futatsugi and members of the ICSE 2008 Program Committee for recognizing the impact of the work behind our ICSE 1998 paper and giving us the opportunity of presenting this paper.

7. REFERENCES

[1] *VMware*. <http://www.vmware.com/>.

[2] *Xen*. <http://xen.org/>.

[3] *The Globus Alliance*. <http://www.globus.org/>.

[4] Allen, R.J., Douence, R., and Garlan, D. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering*. Lisbon, Portugal, March 1998, 1998.

[5] Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. SETI@home: an experiment in public-resource computing. *Communications of the ACM*. 45(11), p. 56-61, November, 2002.

[6] Baset, S.A. and Schulzrinne, H. *An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol*. Columbia University, Report CUCS-039-04, 2004.

[7] Brun, Y. and Medvidovic, N. An Architectural Style for Solving Computationally Intensive Problems on Large

Networks. In *Proceedings of the SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. p. 2-9, IEEE Computer Society. 2007. <http://dx.doi.org/10.1109/SEAMS.2007.4>.

[8] Carzaniga, A., Rosenblum, D.S., and Wolf, A.L. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*. 9(3), p. 332-383, August, 2001.

[9] Chan, A.T.S. and Chuang, S.-N. MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing. *IEEE Transactions on Software Engineering*. 29(12), p. 1072-1085, December, 2003.

[10] Dean, J. and Ghernawat, S. MapReduce: Simplified Data Processing on Large Clusters In *Proceedings of the OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, December, 2004.

[11] Erenkrantz, J.R., Gorlick, M., Suryanarayana, G., and Taylor, R.N. From Representations to Computations: The Evolution of Web Architectures. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Dubrovnik, Croatia, September, 2007.

[12] Eugster, P.T., Felber, P.A., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Computing Surveys*. 35(2), p. 114-131, 2003.

[13] Fielding, R.T. and Taylor, R.N. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology (TOIT)*. 2(2), p. 115-150, May, 2002.

[14] Foster, I., Kesselman, C., Nick, J.M., and Tuecke, S. Grid Services for Distributed System Integration. *IEEE Computer*. 35(6), p. 37-46, June, 2002.

[15] Garlan, D., Siewiorek, D., Smailagic, A., and Steenkiste, P. Project Aura: Toward distraction-free pervasive computing. *IEEE Pervasive computing*. 4(2), p. 22-31, April, 2002.

[16] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*. p. 46-54, October, 2004.

[17] Gorlick, M.M. and Razouk, R.R. Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th International Conference on Software Engineering*. p. 23-34, May, 1991.

[18] Inverardi, P. and Wolf, A.L. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*. 21(4), p. 373-386, April, 1995.

[19] Kramer, J. and Magee, J. Self-Managed Systems: An Architectural Challenge In *Future of Software Engineering 2007* Briand, L. and Wolf, A. eds. IEEE-CS Press, 2007.

[20] Larson, S.M., Snow, C.D., Shirts, M.R., and Pande, V.S. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. In *Computational Genomics*. Horizon Press, 2002.

[21] Luckham, D.C. and Vera, J. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*. 21(9), p. 717-734, September, 1995.

[22] Magee, J. and Kramer, J. Dynamic Structure in Software Architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*. p. 3-14, ACM SIGSOFT. San Francisco, CA, October 16-18, 1996.

[23] Metayer, D.L. Describing software architecture styles using graph grammars. *Transactions on Software Engineering*. 24(7), p. 521-553, July, 1998. <http://www.computer.org/tse/ts1998/e0521abs.htm?SMSESSION=NO>.

[24] MicrosoftTechNet. *Introduction to Hotpatching*. <http://technet2.microsoft.com/windowsserver/en/library/8bf7c6e4-3175-43bd-a67a-827ff3a586011033.mspx?mfr=true>, Microsoft Corporation, 2008.

[25] Ommering, R.v., Linden, F.v.d., Kramer, J., and Magee, J. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*. 33(3), p. 78-85, March, 2000.

[26] Oreizy, P. and Taylor, R.N. On the Role of Software Architectures in Runtime System Reconfiguration. *IEE Proceedings - Software Engineering*. 145(5), p. 137-145, October, 1998.

[27] Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., and Wolf, A.L. An Architecture-based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*. 14(3), p. 54-62, May-June, 1999.

[28] Pingdom. *Best and worst US online banks revealed*. <http://www.pingdom.com/_img/press/best_and_worst_us_online_banks_revealed.pdf>, 2006.

[29] Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*. 7(4), p. 57-66, July, 1990.

[30] Reuters. Routine Upgrade Blamed for BlackBerry Outage. *New York Times*. 12 February, 2008. <http://www.nytimes.com/2008/02/12/technology/12cnd-rim.html?ex=1360558800&en=eceb00610baba273&ei=5124&partner=permalink&exprod=permalink>.

[31] Taylor, R.N., Medvidovic, N., Anderson, K.M., E. James Whitehead, J., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*. 22(6), p. 390-406, June, 1996.

[32] Taylor, R.N., Medvidovic, N., and Dashofy, E.M. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2008. In press.

[33] Winfree, E. *Simulations of computing by self-assembly of DNA*. California Institute of Technology, Report CS-TR:1998:22, 1998.

[34] C. R. Hofmeister. Dynamic Reconfiguration of Distributed Applications. Ph.D. Thesis. University of Maryland, Computer Science Department, 1993.